EE524 HW2: "Hello Parallel World" kernel and Performance Profiling

Problem1:

A. Compute the arithmetic intensity (AI) / operational intensity (OpI) of the following BLAS level 1 (DAXPY) and level 3 (DGEMM) routines.
    a. Refer to lecture 3 slides 30-32 for definitions of AI/OpI and DAXPY/DGEMM. Note the "D" stands for double here.
    b. Show your work (computational work W and memory traffic Q = Qr + Qw) see slide 32.

B. Roofline chart (lecture 3 slide 34)
    a. Determine your computer system's peak floating-point performance (GFlops/s) and peak memory bandwidth (GBytes/sec) by referring to product specifications.
    b. Draw the Roofline chart for your system and add the vertical lines for the AI/OpI values for the 3 BLAS kernels: DAXPY, DGEMV (which we covered in lecture 3), DGEMM.

Problem2:

We'll use the skills you've built with defining simple device-side kernels in OpenCL C and implementing the Host-side OpenCL Platform and Execution model API application "scaffolding" to investigate the parallel kernel execution workgroup and workitems, to experiment with more advanced OpenCL C data types and syntax, and to use profiling to start analysing kernel performance.

Goals:

1. Gain familiarity with
    a. OpenCL NDRange index space workgroup and workitem concepts.
    b. Details of OpenCL C syntax and built-in function usage.
    c. Define user-defined data-types and structures and pass between Host and Device(s)
    d. Using command profiling events and windows Performance counters to analyze and profile application performance.

Procedure:

1. Use the basic OpenCL host application you created in EX2b. This will provide the basic application "scaffolding" we'll use as starting point for almost every OpenCL application.
2. "Hello Parallel World" kernel
    a. Write an NDRange=2D kernel which accepts the following arguments
        i. float3, float4, and float8 vectors
        ii. a pointer to a user defined struct in global memory, including (in this order)
            1. char
            2. char4
            3. union {float f; short s; char c; } u;
            4. array of 4 uint2 vectors
        iii. on the Host side, this structure definition will need to be duplicated, but using the Host API variable types, e.g. cl_char, cl_char4, etc…
            1. this will be used to define the input memory buffer to pass as the kernel argument
    b. use built-in OpenCL C *printf( )* within the kernel to print the following items:
        i. the 2D global ID of each work-item (kernel instance)
            1. use a small index space such as (5,5,0) to keep it manageable
        ii. the 2D local work-item IDs and work-group IDs
        iii. the contents of the float4 vector, element-wise
        iv. the reversed contents of the float4 vector

        v. the "swizzled" high/low and even/odd elements of the float16 vector
        vi. the contents of the structure members
        vii. the sizeof() the entire structure
        viii. the sizeof() the sum of all the individual structure elements
        ix. the sizeof() the structure union element: u
        x. the sizeof() the structure union sub-element: u.c
    c. Modify your host application to launch/enqueue the `helloparallelworld` kernel
        i. Update kernel arguments to match kernel function signature
        ii. Initialize all the variables on the Host side application before invoking kernel.
    d. Try different NDRange variations of the global and local work-sizes:
        i. Such as:
            1. Global = {5,5,0} and local = {5,5,0}
            2. {5,5,0} and {1,1,0}
            3. {5,5,0} and {2,2,0}
        ii. Observe the execution outputs
            1. Note the execution ordering of the work-items with various configurations of global and local work-item sizes.
            2. Ensure the outputs of variable sizes and elements are what you expect.

3. Performance Profiling
    a. We'll use the two (naïve, Opt1) versions of our MMUL kernel.
        i. Make total # of buffer (vector) elements = 262,144
        ii. Make global work-item dimensions {512,512,0}
        iii. Local work-item dimensions: set equal to Best Configuration from KDF
    b. Write 2 distinct profiling loops (with 500 iterations each) which will gather and calculate a cumulative average kernel execution time and the standard deviation using two different profiling methods:
        i. Windows Performance Counter (WPC) API
            1. 2 sub-cases. For each case, put your calls to QueryPerformanceCounter(…) immediately before and after the specified OpenCL API functions:
                a. WPC-Case-1
                    i. clEnqueueNDRangeKernel(…)
                b. WPC-Case-2
                    i. clEnqueueNDRangeKernel(…)
                    ii. clFinish(…)
        ii. OpenCL (OCL) event-based command profiling
            1. We'll measure the difference between 2 sets of 2 profiling info values:
                a. OCL-Set-1:
                    i. CL_PROFILING_COMMAND_SUBMIT
                    ii. CL_PROFLING_COMMAND_END
                b. OCL-Set-2:
                    i. CL_PROFILING_COMMAND_START
                    ii. CL_PROFLING_COMMAND_END
    c. Only gather timing info from one or the other at a time, so they don't interfere with each other's accurate timing information.
    d. The difference between the Windows performance counter timing and the event-based profiling timing is the CPU-GPU communication overhead

Expected output:

- Console output showing `printf( )` statements from the helloparallelworld kernel instances
- Profiling
    - Average and Standard Deviation timing results over 500 iterations of kernel execution for each of following:
        - Windows performance counter WPC-Cases 1, 2
        - OpenCL event profiling OCL-Set 1, 2
    - Discuss differences observed between the 4 cases above. What do you observe?
        - Try this on both GPU and CPU OpenCL devices
        - Do the OpenCL kernel timing results agree with Code Builder KDF Best Configuration execution median values?